Volume Graphics (2005)
E. Gröller, I. Fujishiro (Editors)

Texturing and Hypertexturing of Volumetric Objects

C. M. Miller[†] and M. W. Jones[‡]

Department of Computer Science, University of Wales Swansea, UK

Abstract

Texture mapping is an extremely powerful and flexible tool for adding complex surface detail to an object. This paper introduces a method of surface texturing and hypertexturing complex volumetric objects in real-time. We employ distance field volume representations, texture based volume rendering and procedural texturing techniques with Shader Model 2.0 flexible programmable graphics hardware. We aim to provide a flexible cross-platform, non vendor specific implementation.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture-Graphics Architecture I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling I.3.6 [Computer Graphics]: Methodology and TechniquesGraphics data structures and data types I.3.7 [Computer Graphics]: Three Dimensional Graphics and RealismColor, Shading, Shadowing and Texture I.3.8 [Computer Graphics]: Applications I.4.10 [Image Processing and Computer Vision]: Image RepresentationVolumetric

1. Introduction

Whilst most volume graphics applications concentrate on providing visualizations of data, having tools to compute more general imagery from volumes is also necessary [CKY00]. Texture mapping is commonly used to enhance the appearance of an object by adding complex image information. Methods such as bump mapping [Bli78], environment mapping [BN76] and solid texturing [Pea85] have also been developed to add detail to surfaces without a serious impact on rendering speed. A volume object can also be textured to add visual information at the surface in the same manner [SJ02].

Hypertexture [PH89], a texturing method that is used to create realistic natural looking phenomena in procedural texturing [EMP*02] can be used to create many interesting objects. Hypertexture has been extended to volume graphics [SJ02] in order to give external object detail as well as maintain internal object detail.

These volume texturing approaches (both surface texturing and hypertexturing) are currently far from real-time. The goal of this work is to demonstrate that it is possible to provide the hypertexture effect in real time on complex objects that have been voxelised as a distance field. This paper introduces a method which takes advantage of programmable graphics hardware and distance fields in order to enable real-time texturing and hypertexturing of volumetric objects.

These new real-time techniques can be employed in many fields where external and internal object detail is required. One such application would be the games industry, where effects such as fire, glow and melting are extremely common. Having internal object data with such effects would also provide better playability when deformations occur in game play. They could also be of use for some scientific visualization techniques (e.g. NPR [RE01,TC00]). It's also our intention to expand the rendering techniques available within the volume graphics domain.

Section 2 contains a review of previous work in hardware assisted volume rendering, distance fields, procedural texturing and volume texturing. Section 3 presents the rendering algorithm utilizing the GPU. Section 4 explores alternative methods to procedurally texture on the GPU using noise as a primitive. Section 5 outlines solid volume texturing and bump mapping. Section 6 explains hypertexturing volume objects and finally section 7 outlines testing and results.

[†] e-mail: cschrism@swan.ac.uk
[‡] e-mail: m.w.jones@swan.ac.uk

2. Previous Work

Volume rendering by ray casting is a well established rendering technique in volume graphics [Lev88]. The original ray casting technique has been adapted to work with 3D texturing in order to accelerate the algorithm and take advantage of the parallel nature of today's GPU's [CCF94, CN94, GK96].

Generally hardware methods approximate the volume rendering algorithm, but benefit from the massively parallel GPU fragment processing engine. The principle is to have proxy geometries representing slices in a volume, then use texture mapping hardware to encode the volume information onto each proxy slice. These proxy slices are facing the viewpoint, and when rasterized leave a number of slices of fragments stacked on top of one another (see Figure 1).

After rasterization of the proxy geometry into fragments, each fragment is processed as follows: Firstly the volume texture is queried using the fragments 3D texture position, this value is used in a dependent texture read for classification, usually into a 1D lookup table. This yields the fragments colour and opacity information. After fragment processing alpha blending is used for writing to the frame buffer.

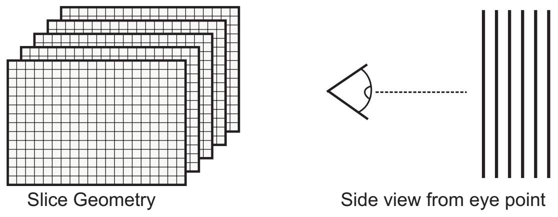


Figure 1: *Parallel projection of slice geometry and side view of geometry.*

Before 3D texturing became a feature of GPU's, 2D textures were used [RSEB*00]. 3D texturing provides tri-linear interpolation in hardware and provides better images. 2D textures also require more than one copy of the proxy geometry for rendering different viewpoints.

There are two approaches to classification. For post-classification, the scalar voxel value is interpolated and then used within a lookup table to obtain colour and opacity. Pre-classification finds the colour and opacity at discrete voxel positions and interpolates colour and opacity from those. Post-classification is regarded as the correct approach and can be achieved in hardware by using tri-linear interpolation within the texturing unit. Pre-integrated classification [EKE01] yields the best results but has the highest impact on rendering time. Post-classification produces comparable results to pre-integrated classification when using low frequency data and transfer functions. For most simple cases post-classification reproduces good enough results at a reduced rendering time. Pre-classification produces the lowest quality results and usually produces blocky unsmoothed final images. We use post-classification in this work.

By knowing the gradient normals for the volume, its also possible to have lighting effects [RSEB*00, EKE01, RGW*03]. Per-pixel lighting can be implemented along with volume rendering in hardware. Over sampling and shadows are also possible [RGW*03], although the reported frame rates are not real-time. The volumes used in their examples are quite large and also classification is done with complex transfer functions. These methods most probably run at real-time on simpler volumes and lower order transfer functions. Over-sampling requires more texture fetches which will slow rendering down. Shadowing requires more lighting computation, and will also slow each fragment down.

2.1. Distance Fields

Distance fields are a volume representation method based on the distance to a chosen surface. Creating a distance field volume involves taking the original surface of interest and restructuring the volume as follows: A distance of 0 is used to denote the surface; positive values are built around the surface approximating the distance to the nearest surface point of the chosen surface and negative values are used to denote internal object distances to the surface. There are various different approaches to generating distance fields [SB02, GPRJ00, Jon96, SJ01]. The distance field is generated for a chosen iso-surface for the object as a preprocessing step.

2.2. Solid Texturing

Solid texturing is an extension to basic 2D texturing that works on a 3D basis [Pea85]. Instead of wrapping or projecting the texture, the objects surface is effectively carved out of the 3D texture map. This method fits best with the volume graphics paradigm since we are usually concerned with a 3D domain. Carving is particularly useful for modeling materials such as wood and marble, as a 3D block of marble or wood is shaped using the objects definition as if the object was shaped from an original block of material (see Figure 2).

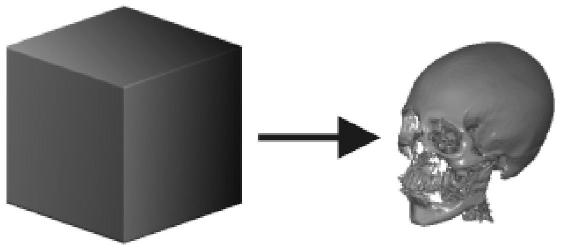


Figure 2: *Solid texture block and solid textured object.*

Satherley and Jones [SJ02] mention solid texturing for volume objects in software. In hardware this is a simple texture lookup instruction from the surface of interest into a 3D texture or procedurally generated texture algorithms can be evaluated for the surface point of interest [CH02]. Many procedural texturing techniques are especially suited to solid texturing.

2.3. Hypertexture

Perlin and Hoffet [PH89] extended the 3D texture mapping paradigm to produce hypertexture, allowing modeling of natural phenomena such as fur, fire and smoke.

Hypertexture is a method of manipulating surface densities whilst rendering, instead of evaluating colouring at the local surface. The actual surface definition is changed during rendering. Because of this rendering outside the conventional surface definition, a ray marcher must be used.

Hypertexture is modelled with an object density function (see Eq. 1), giving rise to the notion of a soft object, or object which has a surface with depth associated. This gives three possible states to define an object:

- Inside - The point is inside object
- Outside - The point is outside the object and soft region
- Boundary - The point is in the soft region or soft surface.

$$D(p) = \begin{cases} 1 & \text{if } |p|^2 \leq r_i^2, \\ 0 & \text{if } |p|^2 \geq r_o^2, \\ \frac{r_o^2 - |p|^2}{r_o^2 - r_i^2} & \text{otherwise.} \end{cases} \tag{1}$$

where r_i = inner radius and r_o = outer radius.

Hypertexture effects can now be achieved by the repeated application of density modulation functions (DMF's) to the soft region of $D(p)$, as shown in Eq. 2.

$$H\big(D(p), p\big) = DMF_n\Big(\ldots\Big(DMF_0\big(D(p)\big)\Big)\Big) \tag{2}$$

DMF's can be defined using noise and turbulence

$$noise(p) = \text{Random scalar} \tag{3}$$

$$turbulence(p) = \sum_i abs\left(\frac{noise(2^i p)}{2^i}\right) \tag{4}$$

Bias and Gain are also introduced:

$$bias_b\big(D(p)\big) = D(p)^{\frac{ln(b)}{ln\frac{1}{2}}} \tag{5}$$

$$gain_g\big(D(p)\big) = \begin{cases} \frac{bias_{1-g}\left(2D(p)\right)}{2} & \text{if } D(p) < \frac{1}{2}, \\ 1 - \frac{bias_{1-g}\left(2-2D(p)\right)}{2} & \text{otherwise.} \end{cases} \tag{6}$$

Perlin later improved on his noise implementation [Per02] to iron out problems with the gradient calculations.

Hart [Har01] implemented Perlin noise on earlier GPU's using NVIDIA's register combinators. Whilst this method is transferable to more modern fragment shaders, the method is not particularly fast, and in large domains will slow the rendering speed down significantly. This is due to the massive multi-pass needs of the algorithm. Since volume rendering on GPU's requires massive fragment processing overheads, the addition of computing noise in a multi-pass manner is too expensive. The method presented here avoids the multi-pass approach in order to maintain a low overhead to the fragment shading unit.

3. Volume Distance Field Rendering

The aim of this work is to demonstrate real-time hypertexturing of complex objects. Previously Satherley and Jones [SJ02] demonstrated a software approach that generated a 300 × 300 image in 50 seconds. This work demonstrates a hardware approach that employs the OpenGL [SA] graphics API. There are numerous high-level shading languages available to program the vertex and fragment processors of today's GPU's. GLSL [KBR] was chosen for the programming of shaders because it integrates well into OpenGL and allows access to the fixed function pipeline variables with relative ease. It also allows multiple program objects to be linked together providing a mechanism to easily switch out specific functions, which cuts down run-time linking and eases swapping in different effects whilst maintaining real-time.

Various Architecture Review Board (ARB) extensions [Ope] to OpenGL are used to access a GLSL implementation. These are `ARB_SHADER_LANGUAGE_100`, `ARB_FRAGMENT_SHADER` and `ARB_VERTEX_SHADER`. We also use the `EXT_TEXTURE_3D` extension for 3D texturing capability. Vendor specific instructions are not used to ensure cross platform compatibility.

3.1. Distance Field Volume Texture

A distance field can be used to define the soft region around an object for hypertexture. Currently, there are extensions to OpenGL to allow arbitrary texture dimensions, as standard OpenGL textures must conform to power of 2 sizes. Arbitrary 32 bit floating point values can also be used to encode texels, as standard OpenGL textures are limited to 8 or 16 bit precision over the $[0,1]$ range. These are exposed though the `NV_FLOAT_BUFFER` and

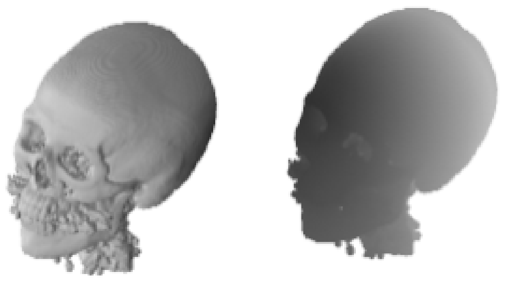


Figure 3: *CT Head gradient normals and texture co-ordinates*

NV_TEXTURE_RECTANGLE extensions. However, currently this extension only supports the GL_NEAREST texture mode, which bypasses the much needed tri-linear interpolation. Therefore in order to use tri-linear interpolation the distance field volume should have a maximum precision of 16 bits and be normalized to the $[0,1]$ range. Also the volume dimensions must be compliant to power of 2 sizes.

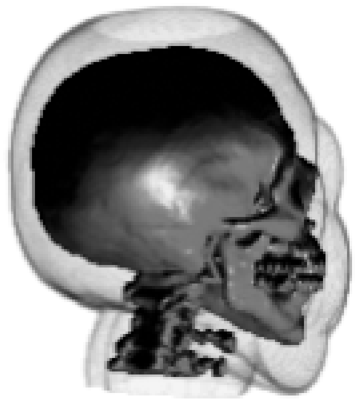


Figure 4: *CT Head with example softregion*

As we wish to do lighting calculations, the gradient normals also have to be present in the texture. Since the densities of a normal Distance Field volume can be put into one channel of a texture, there are 3 more channels in which to put the gradient normal. A good configuration for this is $< r,g,b >=$ gradient normal, $< \alpha >=$ density. See Figure 3(a) for gradient normals encoded as colours.

3.2. Lighting

To perform lighting, a per pixel lighting scheme is needed since no surface detail exists at all in the vertex specific geometry (the view aligned slices). So all lighting is done in the fragment shader. We employ a cut down point lighting model to reduce the instruction count [NVI]. This lighting is done in eye space.

3.3. Matrices Setup

When repositioning the viewpoint, the view aligned slice geometries have to remain in a static position. So to rotate or scale the volume we must apply transforms to the texture co-ordinates alone (object space) – see Figure 3(b). To achieve this, the standard OpenGL transformation is bypassed, and a virtual trackball algorithm [HSH04] is used to provide a view matrix as a uniform variable to the vertex shader. An inverse transpose of this matrix is also needed to transform object space co-ordinates into eye-space, and object space normals into eye-space for lighting.

3.4. Per Vertex Operations

The per-vertex operations are transformations that need to be applied to the incoming variables to the vertex shader. The variables consist of the position in object space, the texture co-ordinates in object space, and the material colour. Since the view aligned slices need to be static, we don't change the ModelViewProjection matrix after setting up the windowing, and transform the object space vertex positions into world space using the ModelViewProjection matrix.

The texture coordinates need to be transformed with any rotation that might have been applied to our texture transform matrix. Similarly the object space vertex position must be transformed into eye space for lighting using the inverse transpose of the texture matrix. These texture coordinates and eye space coordinates are then rasterized by the fixed function rasterizer and handed to the fragment shader as varying inputs.

3.5. Per Fragment Operations

The fragment shader receives a number of variables and constants.

Constants:

- Material colour
- Ambient colour
- Light colour
- Specular power
- Light position
- Object iso-value
- Volume texture
- Inverse transpose of the texturing matrix

Variables:

- Object space texture co-ordinates
- Eye space position

Texture:

- Volume distance value for object position
- Volume gradient normal for object position

Firstly iso-surfacing must be done by deciding fragments that satisfy the iso-value. Other fragments are rejected. In order to ensure under-sampling does not create holes in the iso-surface, values greater than the iso-surface are also considered to close gaps.

The gradient normals must then be re-mapped to the $[-1,1]$ range from $[0,1]$ range. This normal must then be transformed to eye space with the inverse transpose of the texture matrix. The lighting computation must then be applied for the final colour.

3.6. Final Pre Frame Buffer Operations

Before a pixel is written to the frame buffer, the blending operation is computed. The first fragments are written straight into the frame buffer (where the frame buffer is currently empty). Subsequent pixels are then blended using the selected OpenGL blending operation. We employ the Over operator, the `ONE_MINUS_SRC_ALPHA` blending operation. Alpha blending is computed for each subsequent fragment using this operator, until the alpha value is full.

4. GPU Based Noise

There are two approaches to getting noise on the GPU, procedurally computed on the GPU (in the fragment shader) and texture encoded noise. Since noise needs to be evaluated at each pixel this is a per fragment operation. Currently there is no hardware support for noise. Ideally noise should be random through an infinite domain so when constructing complex noise, artifacts are not introduced and patterns appear random.

4.1. Texture Based Noise

Firstly, the size of texture to use is important, since maximal space must be left for other texturing or lookup table needs. A sensible size is around 64^3 for a 3D block of noise. The noise volume does not have to be the same dimensions as the volume data set. We are concerned with 3D noise only, since we want to solid texture and hypertexture 3D volume datasets (see Figure 5(a)). Larger texture sizes will give increased resolution.

One problem with texture based noise lookup tables is that when operating outside the texture co-ordinate range, the noise visibly repeats, instead of remaining random. When texture co-ordinates fall outside the range available, there are problems with borders since the `GL_REPEAT` texture mode must be used to access outside the texture-coordinate range. One way of getting around this is to use the `GL_MIRRORED_REPEAT_ARB` (see Figure 6(b)) texture mode, where image quality is better but it still introduces artifacts at some borders.

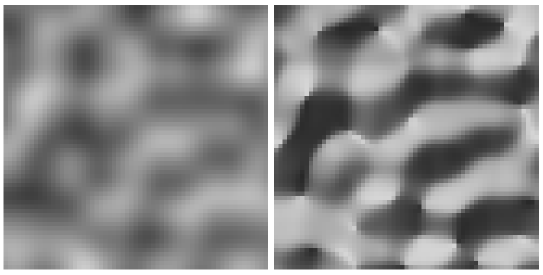

(a) Noise (b) Gradient Normals

Figure 5: *Standard noise and gradient normals*

Pre-computing the noise over a discrete grid in software produces a texture encoded with the final noise configuration. Only 1 texture fetch is needed to utilize this pre-computed noise block (see Figure 6(a)). Therefore texture co-ordinates will not fall outside of range, removing bordering issues. Greater image quality will be generated by using the precision available in software, rendering speed is also good because there is only 1 texture fetch involved.

Computing noise on the GPU using textures is possible via multiple texture fetches into a standard texture lookup table of noise. One channel of a texture can encode a noise block. Higher order noise primitives can then be built up such as turbulence. This method suffers from bordering problems when addressing outside the texture co-ordinate range, and also suffers from not being truly random over an infinite domain. Since the precision on the GPU is lower than in software, the image quality is reduced. It is further reduced by the bordering artifacts just mentioned. Rendering speed is slower because typically more texture fetches are involved, the GPU is also responsible for any additional computation between texture fetches.

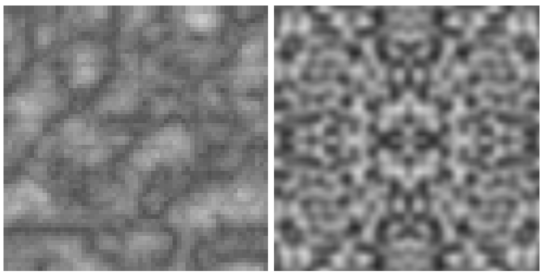

(a) Turbulence (b) Repeating Noise

Figure 6: *8 octave turbulence and repeating noise pattern*

One way of combating the overhead of several texture fetches is to encode different noise blocks in each separate channel of the noise texture. This eliminates 3 additional texture fetches by utilizing the spare channels. The image quality is similar to multiple lookups, but bordering artifacts are reduced. Rendering speed is also increased by reducing the texture fetches.

In some situations, the gradient normals are needed for the noise block (e.g. bump mapping, lighting). By encoding the actual noise density in the alpha channel, the remaining 3 RGB channels can hold the gradient normal (see Figure 5(b)). This requires pre-processing and can be used where only 1 channel is occupied by noise.

It is best to pre-compute the noise block to the required resolution and level of octaves required, this produces best quality and speed.

4.2. Procedural Based Noise

There are two distinct approaches to implementing procedurally based noise on a GPU. The first is encoding the permutation and gradient tables used in the noise computations in a 1D texture lookup table. The second is to use the uniform parameter memory to store these tables as arrays.

By rewriting the noise function in a high level shader language and using texture fetches for permutation and gradient tables, the problem of borders and mirroring using the texture lookup approach can be bypassed, since the original function can be computed. However there are a number of lookups into each table needed, and for standard noise in 3D around 10 texture fetches are needed per fragment. This is too slow for general geometry, but is acceptable for small geometry. Computing 8 octaves of turbulence would thus require 80 texture fetches per fragment, and is clearly too many.

The GLSL spec [KBR] outlines using arrays from uniform variable locations, however current graphics hardware for fragment shaders does not have the ability to do dynamic array addressing with variables, each location must be known at compile time and not run time. However, vertex shaders do have this ability on today's hardware. However since all volume computations require per fragment processing, this method currently cannot be implemented.

5. Surface Texturing Techniques

There are two distinct surface texturing techniques that can be applied to volumes. Solid texturing and bump mapping.

5.1. Solid Texturing

Solid texturing is simple on a GPU. Given the volume rendering of distance fields (see section 3), this is extended by still having the iso-value decision. Instead of using the material colour, an additional texture lookup is done into a block of texture (turbulence, noise etc) with the same fragments texture co-ordinate. This can then be lit using the lighting function or left unlit (see Figure 7(b)).

5.2. Bump Mapping

Instead of using a normal map and computing DOT3 bump mapping, we must combine two different normal maps. This is because we have the surface gradient normal from the volume (recomputed to the $[-1,1]$ range), and we have the gradient normal from the noise texture (which we must also re-map back to the $[-1,1]$ range). These normals must be transformed into eye-space using the inverse transpose of the texture transformation matrix.

In order to perturb the normal, we scale the volume normal with a constant, and add the bump map normal to it (normalized), we then re-normalize the result and use it in the lighting computation. This method is fast, uses few instructions and provides good results. The scaling of the constant affects the severity of the final result (see Eq. 7). Figure 7(c) is an example image of bump mapping.

$$bumpmap(\hat{V},\hat{B},k) = normalize(k\hat{V}+\hat{B}) \qquad (7)$$

where $\hat{V}$ is the volume gradient normal, $\hat{B}$ is the bump map gradient normal and k is the scaling constant.

6. Hypertexturing on the GPU

Hypertexturing complex objects is done by evaluating a surface density function. Section 3 outlines the steps in hardware rendering distance fields, the iso-surfacing method can be modified to implement a surface density function on the fragment shader. We use two iso-values, defining the start of the soft region, and the start of the iso-surface or object (see Figure 3.1). The object is treated in the same manner as the previous explanation of distance field iso-surface rendering.

Differing DMF's applied to the soft region will generate different effects. The construction of several DMF functions (see Eq. 2) can be constructed arbitrarily in the fragment shader.

Usually a DMF will use noise or turbulence as a primitive, and noise must be available to the fragment shader as outlined in section 4. Either procedural or textured noise can be used. The variables available to a DMF at each fragment consist of:

- 3D object position (texture co-ordinates)
- Current volume density (distance to iso-surfaces nearest point)
- Volume Gradient point for current position
- Noise gradient position for current position (if using texture based noise lookups)
- Lighting position
- Eye position

- Material colours
- Arbitrary uniform variables (floats, vectors, matrices) that are constant across fragments (passed into the fragment shader as a uniform parameter) [KBR]
- Standard functions on GPU such as floor, ceil, clamp, sin, cos
- Arbitrary transfer functions encoded as textures

DMF's can be chosen based upon parameters such as spatial co-ordinates, e.g. varying a fire hypertexture effect with height. Different DMF's can be applied through the soft region, although in our examples, the same DMF function was applied throughout the soft region. Lighting can be carried out on the soft region by calling the lighting function resident in the fragment shader from the iso-surface distance field rendering. By using GLSL, a generalized shader can be constructed, and by using the runtime linking and compiling, DMF's can be swapped in and out on the fly. This involves some generalizing of transfer function uniform variables and material properties, but can provide a flexible environment to have interactive shader design tools for volume hypertexturing.

7. Results and Conclusion

The test setup consisted of a Pentium 4 2.8GHz PC. Three graphics cards were tested, the NVIDIA GeForce FX 5900 Ultra, the ATI RADEON 9700 Pro and the NVIDIA GeForce 6800GT. We conducted several tests, basic lit iso-surfacing as described in section 3 (see Figure 7(a)). Solid Texturing (see Figure 7(b)) and bump mapping (see Figure 7(c)) as in section 5. Hypertexture as in section 6 of Fur (see Figure 7(d)), Melting (see Figure 7(e)) and Fire (see Figure 7(f)). Table 1 shows the frame rates achieved with the ATI card for a various window sizes. In general the NVIDIA Geforce FX produced around half the frame rate of the ATI card, although in some situations where texture fetches were heavy, the NVIDIA GeForce FX was slightly in front. We tested each situation with anti-aliasing and antiostropic filtering turned both off, both at 4x and both at 8x. The difference between the 4x and 8x speeds were marginal. The difference in speed between the NVIDIA and ATI cards tested is probably due to the way in which fragments are discarded. The newer NVIDIA GeForce 6800GT unsurprisingly outperformed the older cards due to correct fragment discarding and additional fragment processing power.

We also tested a number of GPU noise techniques as discussed in section 4. We concentrated on turbulence as the detailed texturing method. See Figures 8(c), 8(a), 8(b) and 8(d) respectively for images of the 8 noise lookups, 4 noise lookups, 1 lookup for 4 channels and 1 lookup for pre-computed noise. Table 3 shows the results. We do not report the different turbulence methods for the new NVIDIA card here since we are interested in comparing each technique. We found the 64^3 pre-computed noise volume provided the best speed and resolution balance.

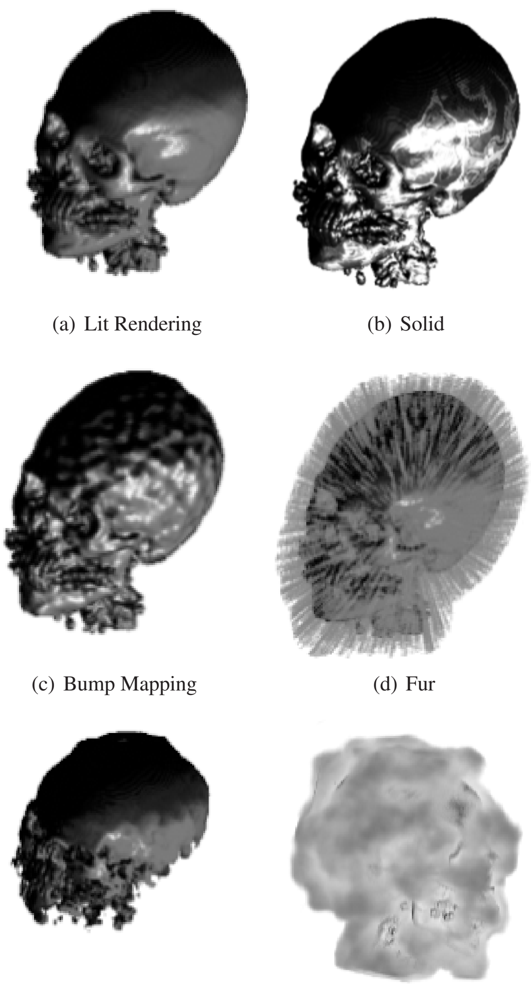

(a) Lit Rendering (b) Solid
(c) Bump Mapping (d) Fur
(e) Melting (f) Fire (bumpy)

Figure 7: *Examples of texturing and hypertexturing an object*

The latest graphics cards (i.e NVIDIA GeForce 6 series) allow iteration in the fragment shader at run-time, which means the original Levoy ray marching approach may be implemented. This sort of implementation will allow space leaping (using the distance field) and will involve less geometry. These new graphics cards also discard fragments correctly and therefore although the processing for each fragment will be much greater than the simple view aligned slice approach, there will be many less fragments to deal with.

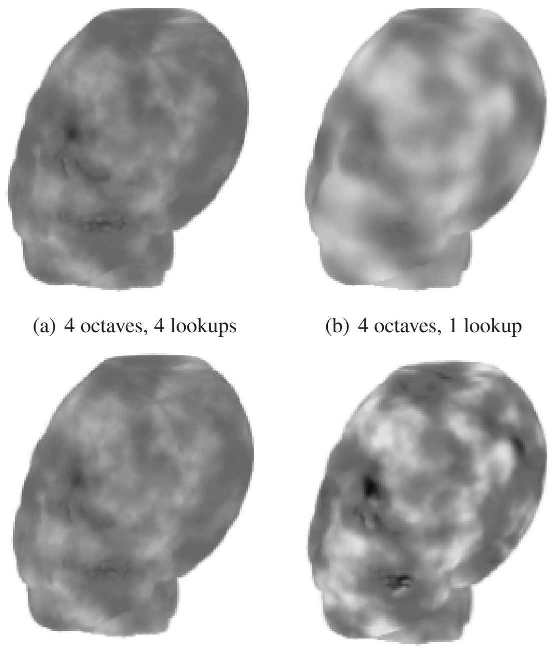
Hardware assisted hypertexturing of complex volume objects extends the available real-time tools for general volume graphics. Applications in volume animation and complex volumetric scenes have been proven to be available on consumer level graphics hardware. The speed of graphics

Table 1: *Frame rates for rendering tests. ATI card*

Window Size	300 × 300		1024 × 768	
Sampling	0x	8x	0x	8x
Lit Rendering	29.82	21.25	5.64	5.01
Solid Texturing	23.21	17.00	4.30	4.26
Bump Mapping	25.26	21.25	4.91	4.72
Fur	17.10	14.17	3.43	3.27
Melting	15.78	10.63	4.02	3.70
Fire	23.24	17.00	4.18	4.05

Table 2: *Frame rates for rendering tests. 6800 card*

Window Size	300 × 300		1024 × 768	
Sampling	0x	8x	0x	8x
Lit Rendering	72.49	46.11	20.57	11.63
Solid Texturing	69.33	45.09	20.02	11.34
Bump Mapping	64.21	41.27	17.52	9.69
Fur	39.52	24.32	9.88	5.43
Melting	57.36	38.26	16.92	9.71
Fire	46.30	27.04	10.42	5.64


(a) 4 octaves, 4 lookups (b) 4 octaves, 1 lookup

(c) 8 octaves, 8 lookup (d) 8 octaves, 1 lookup

Figure 8: *Examples of different turbulence implementations*

hardware is growing at a very fast rate, and the size and complexity of volumes to be rendered will increase drastically in the near future. Already, faster graphics cards are available with more fragment processing power that will cope with larger volumes. With the combination of more per fragment power and a noise implementation on the graphics card itself, more effects on larger volumes will be possible. Hypertexturing could become a very important volume graphics tool in future general graphics applications.

The frame rates achieved here demonstrate how complex texturing and volume rendering can be achieved in real-time with the latest graphics hardware. A full screen 1024 × 768 rendering will run at 10 FPS. This method has been shown to achieve real-time hypertexturing of complex objects for a 300 × 300 image with all cards.

Table 3: *Frame Rates for different turbulence methods. ATI card*

Window Size	300 × 300		1024 × 768	
Sampling	0x	8x	0x	8x
8 noise lookups	2.45	1.89	0.49	0.43
4 noise lookups	7.14	5.26	2.40	2.30
4 channel	23.02	17.02	3.82	3.70
Encoded texture	23.24	17.00	4.18	4.05

References

[Bli78] Blinn J. F.: Simulation of wrinkled surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques* (1978), ACM Press, pp. 286–292.

[BN76] Blinn J. F., Newell M. E.: Texture and reflection in computer generated images. *Commun. ACM 19*, 10 (1976), 542–547.

[CCF94] Cabral B., Cam N., Foran J.: Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of the 1994 symposium on Volume visualization* (1994), ACM Press, pp. 91–98.

[CH02] Carr N. A., Hart J. C.: Meshed atlases for real-time procedural solid texturing. *ACM Trans. Graph. 21*, 2 (2002), 106–131.

[CKY00] Chen M., Kaufman A. E., Yagel R.: *Volume Graphics*. Springer, 2000.

[CN94] Cullip T. J., Neumann U.: *Accelerating Volume Reconstruction With 3D Texture Hardware*. Tech. rep., University of North Carolina at Chapel Hill, 1994.

[EKE01] Engel K., Kraus M., Ertl T.: High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (2001), ACM Press, pp. 9–16.

[EMP*02] EBERT D. S., MUSGRAVE F. K., PEACHEY D., PERLIN K., WORLEY S.: *Texturing and Modelling: A Procedural Approach*, 3rd edition ed. Morgan Kaufmann, 2002.

[GK96] GELDER A. V., KIM K.: Direct volume rendering with shading via three-dimensional textures. In *Proceedings of the 1996 symposium on Volume visualization* (1996), IEEE Press, pp. 23–ff.

[GPRJ00] GIBSON S. F. F., PERRY R. N., ROCKWOOD A. P., JONES T. R.: Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of SIGGRAPH 2000* (2000), pp. 249–254.

[Har01] HART J. C.: Perlin noise pixel shaders. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (2001), ACM Press, pp. 87–94.

[HSH04] HENRIKSEN K., SPORRING J., HORNBÆK K.: Virtual trackballs revisited. *IEEE Transactions on Visualization and Computer Graphics 10*, 2 (2004), 206–216.

[Jon96] JONES M. W.: The production of volume data from triangular meshes using voxelisation. *Computer Graphics Forum 15*, 5 (1996), 311–318.

[KBR] KESSENICH J., BALDWIN D., ROST R.: The opengl shading language. `http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf`.

[Lev88] LEVOY M.: Display of surfaces from volume data. *IEEE Computer Graphics and Applications 8*, 3 (1988), 29–37.

[NVI] NVIDIA: Implementing the fixed-function pipeline using cg. `http://developer.nvidia.com/object/cg_fixed_function.html`.

[Ope] OPENGL: The opengl architecture review board. `http://oss.sgi.com/projects/ogl-sample/registry/`.

[Pea85] PEACHEY D. R.: Solid texturing of complex surfaces. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques* (1985), ACM Press, pp. 279–286.

[Per02] PERLIN K.: Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (2002), ACM Press, pp. 681–682.

[PH89] PERLIN K., HOFFERT E. M.: Hypertexture. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques* (1989), ACM Press, pp. 253–262.

[RE01] RHEINGANS P., EBERT D.: Volume illustration: Nonphotorealistic rendering of volume models. *IEEE Transactions on Visualization and Computer Graphics 7*, 3 (2001), 253–264.

[RGW*03] ROETTGER S., GUTHE S., WEISKOPF D., ERTL T., STRASSER W.: Smart hardware-accelerated volume rendering. In *Proceedings of the symposium on Data visualisation 2003* (2003), Eurographics Association, pp. 231–238.

[RSEB*00] REZK-SALAMA C., ENGEL K., BAUER M., GREINER G., ERTL T.: Interactive volume on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (2000), ACM Press, pp. 109–118.

[SA] SEGAL M., AKELEY K.: The opengl graphics system: A specification (version 1.5).

[SB02] SVENSSON S., BORGEFORS G.: Digital distance transforms in 3D images using information from neighbourhoods up to $5 \times 5 \times 5$. *Computer Vision and Image Understanding 88* (2002), 24–53.

[SJ01] SATHERLEY R., JONES M. W.: Vector-city vector distance transform. *Computer Vision and Image Understanding 82*, 3 (2001), 238–254.

[SJ02] SATHERLEY R., JONES M. W.: Hypertexturing complex volume objects. *The Visual Computer 18*, 4 (June 2002), 226–235.

[TC00] TREAVETT S. M. F., CHEN M.: Pen-and-ink rendering in volume visualisation. In *VIS '00: Proceedings of the conference on Visualization '00* (2000), IEEE Computer Society Press, pp. 203–210.